University of Michigan – Dearborn
Department of Computer and Information Science
CIS 350: Data Structures and Algorithm Analysis
(Winter 2023)
Instructor: Luis E. Ortiz

Program 1: AVL Tree Map with Subtree Statistics Information

Out*: Friday Feb 17, 2023;* **Due***: 2:00pm Thursday Mar 16, 2023*

Overview

- In the *source code* file provided:
  - complete the missing subroutines for performing operations related to the erase method of the (ordered) map ADT implemented using a general BST (BSTMap class)
  - complete the missing subroutines for performing restructuring/rebalance operations in the tree for a map ADT implemented using an AVL Tree (AVLTreeMap class, which extends the BSTMap class)
  - complete the missing subroutines for performing operations required by another implementation of the map ADT (TreeMapStats class) that extends a node data structure in an AVL tree to include and maintin statistics/information about the value of the map related to all the entries stored in the subtree rooted by that node
- In the *memo* file (to be provided separately):
  - provide a worst-case running time and space, using big-Oh notation as function of the maximum number of nodes n in the BST, which roughly characterizes the input size of certain methods used in the implementation

Program Implementation

In class, we learned about self-balancing binary search trees (BSTs). These data structures have the same properties as a regular BST, but with the added benefit that the height of the tree doesn't become too high. This makes self-balancing BSTs more useful in practice than regular BSTs, because operations that manipulate BSTs take longer to run as the tree becomes taller. In this program we will be using one such self-balancing BST data structure, the AVL tree, to implement a map between integer keys and integer values that extends the traditional AVL node data structure to also account for statistics/information related to the map values stored in each subtree rooted at the node.

There are three main components for this program. The first is the BST itself, along with the operations needed to make it work. The role of the BST-related class (`BSTMap`) is to provide the basic functionality of a (general, linked-structured) BST needed to implement a map ADT. Some of that functionailty is already provided for you. Your job is to implement some missing functions needed for the proper functionality of the erase operation of the map.

The second component involves the AVL Tree. The class related to the AVL tree (`AVLTreeMap`) inherits from the BST class, so, once the first component is completed, it will have all the functionality of a BST-based map available. As we learned in lectures, inserting and deleting in an AVL tree are the same as in a BST, but with some added steps at the end where we have to check the tree's balance and perform a rotation operation if necessary. Your job is to implement the function to rebalance an unbalanced node and to perform single-rotation restructuring. This will necessary include mantaining the consistency of the AVL node class (i.e, to maintain the consistency of the

height attribute `ht` of the embeded class `Node` that inherits from the corresponding BST node class embedded within `BSTMap`).

The third component involves an extension of the AVL Tree to account for additional statistics/information stored at its nodes for the implementation of a map ADT. The class related to this AVL Tree extension (`TreeStatsMap`) inherits from the AVL Tree class, so, once the first two components are completed, it will have all the functionality of an AVL-Tree-based map available. For this project, the information maintained for each *subtree* is the *number of nodes*, as well as the *sum*, *minimum* and *maximum* of the *map values* of the entries stored in the subtree. An embeded class (`Stats`) of an extension of the AVL node class (`Node`), which is itself embedded within the AVL Tree extension class (`TreeStatsMap`), has data members to store that statistics/information within a data member (`info`) extending the node class (`Node`) for the AVL extension class (`TreeStatsMap`). Your job is to implement functions to maintain the cosistency of the required statistics/information stored at each node.

Keep in mind that **adding functions** to aid your implementations is **perfectly acceptable** (and may make things much easier in some cases). However, **do not** change any of the code already provided or add extra input/output to the program, as that may interfere with the grading process. it is **your responsability** to remove or disable any extra input/output code that you might have added during your debugging process before uploading your submission to the autograder.

Source Code Details

We provide the source code for the linked-structure BST, AVL tree, and its extension for a map implementation in three languages: C++, Java, and Python. This program provides class definitions and basic functionality needed to construct the relevant data structures for your algorithm implementations and to handle input/output. However, the source will be missing the implementation for some important functions.

We expect you to complete the following functions in the `BSTMap` class:

- `eraseNode`: Perform a standard BST node deletion for a key in the map.

We expect you to *complete* the following *functions* in the `AVLTreeMap` class:

- `rebalance`: Given an unbalanced subtree rooted at Node z, balance the subtree using a single or double rotation as needed.
- `singleRotation`: Perform a single rotation operation on the subtree rooted at Node z such that Node y becomes the new root of the subtree.

Also *complete* the following *functions* in the `TreeMapStats` class:

- `putNode`: Insert or update the entry in the extended AVL tree map (`TreeMapStats`) using the inputted key and value. In addition, efficiently update the statistics information of only the nodes affected by the insertion of the new entry into the map
- `eraseNode`: Remove the entry in the extended AVL tree map (`TreeMapStats`) with the inputted key, if the key is in the map, if necessary. In addition, efficiently update the statistics information of only the nodes affected by the removal of the new entry from the map

- `singleRotation`: Perform a single rotation on the input nodes `y` and (its parent) `z` and update the statistics/information as needed to maintain consistency

NOTE: Do not forget to write your name in the source code file.

Input File Format and Sample File

Use a file named `input.txt`, which is assumed to be located in the same subdirectory where the program is executed, to debug, test, and evaluate your implementation.

Input File Format

Each line in the input file can represent a single or multiple commands depending on the context.

- `put <key> <value>`

  Adds or updates a map entry with the specified key and value.

- `erase <key>`

  Removes from the map the entry with the specified key, if it exists.

- `find <key>`

  Prints out the value associated with the key if in the map; otherwise, prints out "No value!"

- `print_key_stats <key>`

  Prints out all the information associated with the node in the tree storing the entry with the given key, if in the map, as

  `<key>:<value>(<height>){<num>,<sum>,<min>,<max>}`

  where

  - `<key>` is the *key* of the *entry* stored in the node,
  - `<value>` is the *value* of the *entry* stored in the node,
  - `<height>` is the *height* of the *subtree* rooted at the node of AVL tree where the entry is stored,
  - `<num>` is the *size* of the *subtree* (i.e., the number of nodes/entries stored) rooted at the node of the AVL tree where the entry is stored,
  - `<sum>` is the *total sum* of all the *map values* of the entries stored in the *subtree* rooted at the node of the AVL tree where the entry is stored,
  - `<min>` is the *minimum* of all the *map values* of the entries stored in the *subtree* rooted at the node of the AVL tree where the entry is stored, and
  - `<max>` is the *maximum* of all the *map values* of the entries stored in the *subtree* rooted at the node of the AVL tree where the entry is stored.

  Otherwise, if the key is not in the map, it prints out "No value!"

- `size <size>`

  Prints out the number of entries in the map.

- `print`

  Display the AVL tree *map entries* using a preorder/infix parenthetic representation:
  `[<key>:<value>](<left_tree>),(<right_tree>)`

- `print_tree`

  Display the AVL tree *map entries* using a string representation which reflects the structure of the tree:

  ```
            right_child key:value
  root key:value
            left_child key:value
  ```

- `print_stats`

  Display the AVL tree *node statistics/information* using a preorder/infix parenthetic representation:
  `[<key>:<value>(<height>){<num>,<sum>,<min>,<max>}](<left_tree>),(<right_tree>)`

- `print_stats_tree`

  Display the AVL tree *node statistics/information* using a string representation which reflects the structure of the tree:

  ```
            right_child key:value(height){num,sum,min,max}
  root key:value(height){num,sum,min,max}
            left_child key:value(height){num,sum,min,max}
  ```

To be provided/discussed separately:

- sample input file `input.txt`, and its corresponding sample output
- the memo file and how to submit it
- details on how to submit the program source code
- details on how the program will be graded (e.g., time/space analysis, functionality, and style)